UNIVERSIDADE FEDERAL DA FRONTEIRA SUL
CAMPUS DE CHAPECÓ
CURSO DE CIÊNCIA DA COMPUTAÇÃO

FELIPE CHABATURA NETO

O SISTEMA OPERACIONAL EM TEMPO REAL FREERTOS NA PLATAFORMA DE HARDWARE ARDUINO UNO

UMA ANÁLISE DOS RECURSOS DE GERENCIAMENTO DE MEMÓRIA, TAREFAS, FILAS E DO DESEMPENHO GERAL DO SISTEMA

CHAPECÓ
2018

FELIPE CHABATURA NETO

O SISTEMA OPERACIONAL EM TEMPO REAL FREERTOS NA PLATAFORMA DE HARDWARE ARDUINO UNO

UMA ANÁLISE DOS RECURSOS DE GERENCIAMENTO DE MEMÓRIA, TAREFAS, FILAS E DO DESEMPENHO GERAL DO SISTEMA

Trabalho de conclusão de curso apresentado como requisito para obtenção do grau de Bacharel em Ciência da Computação da Universidade Federal da Fronteira Sul.
Orientador: Prof. Dr. Marco Aurélio Spohn

CHAPECÓ
2018

Neto, Felipe Chabatura

O SISTEMA OPERACIONAL EM TEMPO REAL FREERTOS NA PLATAFORMA DE HARDWARE ARDUINO UNO / Felipe Chabatura Neto. – 2018.
47 f.: il.

Orientador: Prof. Dr. Marco Aurélio Spohn.
Trabalho de conclusão de curso (graduação) – Universidade Federal da Fronteira Sul, curso de Ciência da Computação, Chapecó, SC, 2018.

1. Sistemas Operacionais em Tempo Real. 2. Avaliação de Desempenho. 3. FreeRTOS. 4. Arduino. I. Spohn, Prof. Dr. Marco Aurélio, orientador. II. Universidade Federal da Fronteira Sul. III. Título.

E-mail: felipechabat@gmail.com

FELIPE CHABATURA NETO

O SISTEMA OPERACIONAL EM TEMPO REAL FREERTOS NA PLATAFORMA DE HARDWARE ARDUINO UNO

UMA ANÁLISE DOS RECURSOS DE GERENCIAMENTO DE MEMÓRIA, TAREFAS, FILAS E DO DESEMPENHO GERAL DO SISTEMA

Trabalho de conclusão de curso apresentado como requisito para obtenção do grau de Bacharel em Ciência da Computação da Universidade Federal da Fronteira Sul.

Orientador: Prof. Dr. Marco Aurélio Spohn

Este trabalho de conclusão de curso foi defendido e aprovado pela banca avaliadora em: 6/12/2018.

BANCA AVALIADORA

Marco A. Spohn

Prof. Dr. Marco Aurélio Spohn – UFFS

Prof. Dr. Emílio Wuerges – UFFS

Adriano S. Padilha

Prof. Me. Adriano Sanick Padilha – UFFS

“You may delay, but time will not.”

Benjamin Franklin

AGRADECIMENTOS

Em primeiro lugar, eu gostaria de agradecer à minha família, por todo amor e dedicação sem os quais eu não seria grande parte da pessoa que eu sou hoje. Gostaria de agradecer também, ao meu amigo e colega de trabalho Leonardo Tironi Fassini e ao meu orientador Prof. Dr. Marco Aurélio Spohn. Sem a ajuda e dedicação deles, esse trabalho não seria possível. Um muito obrigado à todos os meus amigos pela companhia, ajuda, motivação e inspiração nesses últimos anos. Um agradecimento especial às pessoas que me ajudaram a revisar esse texto: Amadeus Dabela, Guilherme Konopatzki, Jefferson Chabatura e Jardel Holub. Finalmente, agradeço a Alan Turing (1912 - 1954), ateu e homossexual, pai da Ciência da Computação.

RESUMO

A avaliação de desempenho tem papel fundamental quando se é necessário ter medidas quantitativas para comparar diversos sistemas operacionais. Quando se fala em sistemas operacionais em tempo real, geralmente envolvidos em aplicações e sistemas críticos, a situação é ainda mais delicada. Esse trabalho apresenta uma análise da escalabilidade e desempenho dos componentes de gerenciamento de memória, filas e tarefas do sistema operacional em tempo real FreeRTOS na plataforma de *hardware* Arduino Uno. Além disso, faz-se uma avaliação da velocidade de troca de contexto e da comunicação entre tarefas. Os resultados são apresentados numa série de tabelas.

ABSTRACT

Performance evaluation plays a key role when it is necessary to have quantitative measures to compare various operating systems. When you talk about real-time operating systems, usually involved in critical applications and critical systems, the situation is even more delicate. This paper presents an analysis of the scalability and performance of memory management, queues, and tasks components of the FreeRTOS real-time operating system on the Arduino Uno hardware platform. In addition, an evaluation of the speed of task switching and communication between tasks is done. The results are presented in a series of tables.

LISTA DE ILUSTRAÇÕES

Figura 1 – A placa Arduino Uno R3. 26
Figura 2 – Teste para troca de contexto no FreeRTOS. 30
Figura 3 – Teste para comunicação entre tarefas usando filas e notificações de tarefas no FreeRTOS. 31
Figura 4 – Mapa do layout da memória RAM, adotado de (AVR-Libc, 2016). 34

LISTA DE TABELAS

Tabela 1 – Algumas das especificações técnicas do Arduino UNO 26
Tabela 2 – Principais campos do TCB de cada tarefa. 37
Tabela 3 – Tamanho máximo de pilha alocável por tarefa de acordo com a quantidade de tarefas. 40
Tabela 4 – Principais campos da estrutura de fila. 41
Tabela 5 – Máximo de filas por quantidade de tarefas. 43
Tabela 6 – Latência média da troca de contexto do FreeRTOS no Arduino Uno. 43
Tabela 7 – A duração média da transferência de mensagens usando filas. 44
Tabela 8 – A duração média da transferência de mensagens usando notificações de tarefas. 44

SUMÁRIO

1 INTRODUÇÃO 17
1.1 OBJETIVOS 17
1.1.1 Objetivo Geral 17
1.1.2 Objetivos Específicos 18
2 METODOLOGIA 19
3 REVISÃO BIBLIOGRÁFICA 21
3.1 SISTEMAS OPERACIONAIS EM TEMPO REAL 21
3.2 O FREERTOS 21
3.2.1 Tarefas 22
3.2.1.1 Escalonamento de Tarefas 23
3.2.1.2 Notificações de Tarefa 23
3.2.2 Filas 24
3.2.3 Recursos de Memória 24
3.3 O ARDUINO UNO 25
3.3.1 Ambientes de desenvolvimento 26
4 AVALIAÇÃO DE DESEMPENHO 29
4.1 LATÊNCIA MÉDIA DA TROCA DE CONTEXTO ENTRE TAREFAS . . 29
4.2 LATÊNCIA MÉDIA DA TROCA DE MENSAGENS ENTRE TAREFAS . 30
5 RESULTADOS 33
5.1 DOCUMENTAÇÃO AUXILIAR 33
5.2 GERENCIAMENTO DE MEMÓRIA 33
5.2.1 Regiões da Memória RAM 33
5.2.2 Funcionamento da Alocação Dinâmica 34
5.2.3 Como o FreeRTOS lida com alocações dinâmicas 35
5.3 TAREFAS 37
5.3.1 Estrutura do TCB 37
5.3.2 Criação de Tarefas 38
5.3.3 Máximo alocável de tarefas 39
5.4 FILAS 40
5.4.1 Estrutura da fila 40
5.4.2 Criação de Filas 42
5.4.3 Máximo de filas por quantidades de tarefa 43
5.5 TESTES DE DESEMPENHO 43
5.5.1 Latência Média da Troca de Contexto Entre Tarefas 43
5.5.2 Latência Média da Troca de Mensagens Entre Tarefas 44
6 CONSIDERAÇÕES FINAIS 45
6.1 CONCLUSÃO 45

6.2 TRABALHOS FUTUROS 45
REFERÊNCIAS 47

1 INTRODUÇÃO

Atualmente, existe uma grande quantidade de sistemas que possuem requisitos de tempo real. Por exemplo, o controle das turbinas e dos motores de um avião depende de uma série de informações que são coletadas constantemente durante o voo. De maneira análoga, o sistema de acionamento do *airbag* de um carro precisa ser extremamente pontual para que possa ser benéfico ao motorista caso ocorra um acidente. Para suprir essas necessidades, existem os sistemas operacionais em tempo real (RTOS[1]), nos quais se desenvolvem aplicações que cumpram os prazos estipulados pelos cenários que exigem uma resposta dentro de um intervalo de tempo determinado (Rammig et al., 2009).

Quando se projeta uma aplicação em tempo real, torna-se necessário tomar decisões considerando todos os aspectos do *hardware* e do *software* empregados. Para isso deve-se conhecer requisitos da aplicação tais como capacidade de processamento e armazenamento, a fim de não escolher dispositivos demasiadamente potentes ou que não sejam potentes o suficiente para que a aplicação cumpra os requisitos necessários, evitando assim o desperdício de tempo e recursos. Para tomar tal decisão, torna-se necessário que previamente tenha sido feita uma análise, com o intuito de descobrir as capacidades e os limites da combinação de *software* e *hardware* utilizados (Rammig et al., 2009).

Este trabalho propõe a análise de componentes do sistema operacional em tempo real FreeRTOS, um *kernel*[2] em tempo real gratuito e de código livre, na plataforma de *hardware* Arduino Uno, uma placa minimalista e de baixo custo. O port do FreeRTOS aqui analisado não pertence à distribuição oficial, considerando-se que essa versão foi elaborada à parte por Phillip Stevens e, posteriormente, disponibilizada em um repositório no *github* (Stevens, 2018). A análise corrente se dá através do estudo dos componentes do FreeRTOS e suas implementações, bem como auxiliada pela elaboração de testes que exploram os limites destes componentes no ambiente bastante limitado que o Arduino Uno proporciona.

1.1 OBJETIVOS

1.1.1 Objetivo Geral

O objetivo desse projeto é analisar os componentes de gerenciamento de memória, tarefas e filas do sistema operacional em tempo real FreeRTOS na plataforma de *hardware* minimalista Arduino Uno, a fim descobrir os seus respectivos impactos na escalabilidade e desempenho de aplicações no sistema, assim como ter uma noção geral do desempenho do sistema.

---

[1] *Real Time Operating System*.

[2] *Kernel* é o programa principal de um sistema operacional, possuindo controle sobre todo sistema.

1.1.2 Objetivos Específicos

- Analisar o funcionamento do gerenciador de memória presente no *port* do FreeRTOS para Arduino Uno e avaliar o seu impacto no sistema.
- Analisar os recursos de filas e tarefas e suas respectivas escalabilidades dentro de aplicações no sistema.
- Propor e realizar testes para obter medidas quantitativas de desempenho do sistema operacional.
- Criar uma documentação auxiliar que permita melhor entendimento da implementação do sistema operacional.

2 METODOLOGIA

O presente trabalho se iniciou com o estudo do RTOS a ser analisado, em especial dos componentes enfatizados por este projeto, a fim de entender melhor o funcionamento e o propósito de cada um deles dentro do sistema operacional. Inicialmente, esse estudo se deu principalmente pela revisão do guia escrito pelo seu fundador, Richard Barry: "*Mastering the FreeRTOS Real Time Kernel: A Hands-On Tutorial Guide*"(Barry, 2016).

Depois, um estudo exploratório da implementação do *port* do sistema para o Arduino Uno foi conduzido, através da leitura de códigos-fonte. Este estudo teve como propósito obter conhecimento técnico mais aprofundado sobre o funcionamento de cada um dos recursos a serem analisados. Durante essa etapa, uma documentação auxiliar foi produzida, descrevendo os principais tipos, estruturas e funções da implementação do FreeRTOS para Arduino Uno, com o intuito de permitir melhor análise e compreensão do sistema operacional. Essa documentação pode ser acessada através das referências do presente trabalho (Chabatura; Tironi Fassini, 2018).

Para a avaliação do gerenciador de memória, consultou-se a documentação oficial da biblioteca à qual suas funcionalidades pertencem (AVR-Libc, 2016). A partir do estudo do funcionamento das funções que compõe o gerenciador de memória e de como a implementação do FreeRTOS as encapsula, foi possível criar uma análise sobre as consequências do uso desse recurso no desempenho e escalabilidade do sistema.

Para os recursos de tarefas e filas, obteve-se o tamanho de cada estrutura e dos campos que as definem. Dessa forma, é possível saber com determinada precisão a escalabilidade dessas funcionalidades no sistema. Além disso, foram realizados testes para se medir as capacidades máximas de alocação para cada um desses recursos.

Então, para se obter medidas quantitativas do desempenho do sistema operacional, uma revisão bibliográfica na área de análise de desempenho de sistemas operacionais em tempo real foi realizada. Optou-se por usar um subconjunto dos testes realizados por Krzysztof M. Sacha, em seu trabalho "Measuring the Real-Time Operating System Performance"(Sacha, 1995). Os testes realizados tiveram como objetivo medir a velocidade de troca de contexto de tarefas e a velocidade de comunicação usando os recursos de filas e notificações de tarefa.

Por fim, após a realização dos testes e coleta dos dados, se procedeu à análise dos resultados.

3 REVISÃO BIBLIOGRÁFICA

3.1 SISTEMAS OPERACIONAIS EM TEMPO REAL

Um sistema operacional em tempo real é um sistema operacional que garante que determinadas tarefas serão concluídas dentro de limites de tempo determinados, intitulados requisitos de tempo real da aplicação. Ou seja, a corretude do sistema não depende apenas dos resultados que ele produz, mas do tempo em que esses resultados são produzidos. Todos os prazos estipulados devem ser cumpridos, independente das circunstâncias, o que atribui aos RTOS a característica de possuir previsibilidade. Garantir essa previsibilidade pode inclusive tornar o sistema mais lento (Rammig et al., 2009).

Os sistemas em tempo real podem ser divididos pela rigidez de seus requisitos de tempo real:

- *Hard real time requirements* são aqueles que se não alcançados, inutilizam totalmente o sistema e causam consequências catastróficas. Exemplos desses sistemas são sistemas de controle motores de aeronaves e os sistemas que controlam o acionamento do airbag dos carros.

- *Firm real time requirements* são aqueles que se não alcançados, tornam o resultado inútil, mas não comprometem todo o sistema. Como exemplo pode-se citar sistemas de previsão de tempo e os sistemas de decisões do mercado de ações da bolsa de valores.

- *Soft real time requirements* são aqueles que se não alcançados, não causam grave dano, apenas diminuem a performance do sistema. Como exemplo, pode-se citar aplicações de transmissão multimídia e aplicações de comunicação, onde perder uma certa porcentagem dos dados vai gerar ruído ou desconforto, mas não vai tornar a aplicação totalmente inutilizável.

3.2 O FREERTOS

Essa sessão traz uma breve revisão bibliográfica sobre as distribuições do FreeRTOS e sobre as principais funcionalidades do sistema que são abordadas no presente trabalho. Todas as informações a respeito do FreeRTOS foram retiradas do guia oficial: "Mastering the FreeRTOS Real Time Kernel: A Hands-On Tutorial Guide", escrito por seu fundador, *Richard Barry* (Barry, 2016).

O FreeRTOS é um kernel em tempo real gratuito e de código livre, desenvolvido na linguagem C. Ele serve como base para o desenvolvimento de aplicações que precisam cumprir requisitos de tempo real. Segundo seu fundador, ele é idealmente adequado para aplicações profundamente embarcadas que geralmente usam microcontroladores ou pequenos microprocessadores e possuem uma mistura de *soft real time requirements* e *hard real time requirements*.

A distribuição oficial do FreeRTOS possui uma série de arquivos que são compartilhados por todas as combinações de compiladores e arquiteturas para qual o sistema está disponível, além de outros que são específicos para cada conjunto de compilador e arquitetura. Os principais arquivos das distribuições do FreeRTOS são os seguintes:

- ***tasks.c*** - Sempre necessário.
- ***list.c*** - Sempre necessário.
- ***queue.c*** - Quase sempre necessário.
- ***timers.c*** - Opcional.
- ***event_groups.c*** - Opcional.
- ***croutine.c*** - Opcional.

Atualmente, o FreeRTOS está na versão 10.1.1-1. Este projeto foi realizado tomando como objeto de estudo a versão $8.2.3$ do sistema. Entre as diversas correções e melhorias que aconteceram até a versão atual, destaca-se que desde a versão $9.0.0$ o FreeRTOS suporta alocação estática de alguns dos objetos do *kernel*, tais como tarefas, filas, semáforos e *timers*.

Versões do FreeRTOS anteriores à $9.0.0$, como é o caso da versão utilizada nesse projeto, devem também incluir um gerenciador de memória. No caso do *port* para Arduino Uno, apenas um esquema de gerenciamento de memória está disponível, ele é implementado no arquivo *heap_3.c*. Maiores detalhes sobre os esquemas de gerenciamento de memória são fornecidos na sessão 3.2.3

O *port* do FreeRTOS para o Arduino Uno não faz parte da distribuição oficial, ele foi desenvolvido por Phillip Stevens, que o desponibilizou em um repositório público em seu *github* (Stevens, 2018). Mais tarde, o *port* foi disponibilizado nas bibliotecas da plataforma Arduino.

3.2.1 Tarefas

No FreeRTOS, as aplicações são organizadas como coleções de *threads*[1] de execução que são independentes entre si, chamadas de tarefas. Cabe ao *kernel*, mais especificamente ao escalonador, decidir qual tarefa executará de acordo com as prioridades definidas pelo desenvolvedor. No caso de um processador ou microcontrolador que possui apenas um núcleo (como é o caso do *ATmega328P*, no Arduino Uno) apenas uma tarefa executará por vez.

Para cada tarefa, na memória, são alocados uma pilha e um TCB (*Task Control Block* ou bloco de controle da tarefa) onde ficam armazenadas informações de controle da tarefa tais como a prioridade da tarefa, um ponteiro para o início da pilha, entre outras.

Existem vários estados nos quais uma tarefa pode estar em um dado momento de execução da aplicação, são eles: executando, pronta, suspensa e bloqueada. Quando a tarefa está

[1] *Threads* são as menores sequências de instruções programadas com as quais o escalonador lida. São as linhas de execução da aplicação.

executando, ela está utilizando o processador e outros recursos do sistema para executar suas instruções. Tarefas que estão no estado "pronto", estão disponíveis para o escalonador, mas não estão sendo executadas. No estado de bloqueada, a tarefa não está disponível para o escalonador. Neste caso, ela pode estar bloqueada por um evento temporal, tal como esperar uma determinada quantidade de tempo, ou um evento de sincronização, como esperar uma mensagem chegar numa fila. No estado de suspensa, a tarefa também não está disponível para o escalonador, mas só pode ser colocada e retirada desse estado através de chamadas de uma função específica da API (*Application programming interface* ou interface de programação de aplicações).

3.2.1.1 Escalonamento de Tarefas

O FreeRTOS possui diversos algoritmos de escalonamento que variam de acordo com configurações para existência ou não de preempção e fatiamento de tempo. Caso a preempção esteja ativada, o escalonador irá imediatamente trocar a tarefa que está executando atualmente caso outra tarefa com maior prioridade fique disponível. Isso significa que a tarefa que estava sendo executada é movida para o estado de "pronto"involuntariamente (sem ceder lugar nem ficar bloqueada por outro motivo). Esse cenário é conhecido como troca de contexto. O fatiamento de tempo se diz respeito ao compartilhamento de tempo entre tarefas com a mesma prioridade. A cada intervalo de tempo predeterminado, o escalonador irá trocar a tarefa que está executando contanto que haja outra de mesma prioridade no estado de pronta. A essa intervenção do escalonador é dada o nome de interrupção de relógio, uma vez que acontece após o decorrer de uma fatia de tempo. O algoritmo mais comum de escalonamento é o que combina preempção e fatiamento de tempo.

3.2.1.2 Notificações de Tarefa

Notificações de tarefa é um mecanismo de comunicação direta entre tarefas, que elimina a necessidade de uma estrutura intermediária, como uma fila. Nesse recurso, cada tarefa tem um valor de notificação, que consiste de um inteiro de 32 *bits*. Além disso, cada tarefa tem um estado de notificação, que pode ser um dos seguintes valores:

- ***eNotWaitingNotification***: A tarefa não está esperando uma notificação.
- ***eWaitingNotification***: A tarefa está esperando uma notificação.
- ***eNotified***: A tarefa recebeu uma notificação.

Ao notificar uma tarefa, diversas opções podem ser especificadas para se alterar o valor de notificação da tarefa alvo, são elas:

- ***eNoAction***: A tarefa recebe o evento da noticação, mas o valor de notificação não é alterado.

- ***eSetBits***: Ativa alguns bits do valor de noticação através de uma operação OR.
- ***eIncrement***: O valor de notificação da tarefa alvo é incrementado em um.
- ***eSetValueWithOverwrite***: Atribui ao valor de notificação da tarefa alvo o valor passado como parâmetro, incondicionalmente.
- ***eSetValueWithoutOverwrite***: Só atribui ao valor de notificação da tarefa alvo o valor passado como parâmetro caso a tarefa não tenha uma notificação pendente. Caso contrário, a notificação falha e não surte efeito.

Por não precisar alocar um elemento intermediário, o recurso de notificações de tarefas tem um consumo consideravelmente menor de memória RAM e também é significativamente mais rápido. Entretanto, este recurso possui uma série de limitações que o fazem não ser o meio de comunicação mais apropriado em alguns casos. Dentre as principais limitações estão: Não é possível enviar eventos e dados para um serviço de interrupção, não é possível notificar mais de uma tarefa por vez, não é possível armazenar vários dados, como em um *buffer* e também não é possível que uma tarefa espere no estado de bloqueada até que outra que tem uma notificação pendente fique disponível para ser notificada.

3.2.2 Filas

Filas são um recurso extremamente versátil e amplamente usado por todo o sistema operacional e aplicações no geral. Elas proveem um mecanismo de comunicação entre duas tarefas, ou tarefas e serviços de interrupção (serviços de interrupção se referem ao tratamento de eventos que acontecem no ambiente onde o sistema atua, como leituras vindo de um sensor, comandos vindos de um teclado ou pacotes chegando por um periférico de rede).

Filas são normalmente usadas como *buffers* FIFO (*first-in first-out*), onde os dados são escritos no final e lidos do começo. O FreeRTOS provê uma série de recursos para que filas possam ser utilizadas por diversas tarefas, tanto para leitura quanto para escrita. Um exemplo de utilização de uma fila na comunicação dentro do sistema, é uma fila onde uma tarefa escreve valores recebidos pelo serviço de interrupção de um sensor que capta a umidade relativa do ar e outra tarefa aguarda por essas escritas para que possa realizar a leitura desses valores e realizar o processamento desejado.

3.2.3 Recursos de Memória

Segundo o site da companhia do Arduino (AG, 2018), existem três tipos de memória em placas Arduino AVR, como é o caso do Arduino Uno. São elas:

- Memória Flash: É onde as instruções do programa ficam armazenadas.

- Memória SRAM[2]: É onde ficam os dados durante a execução do programa. Variáveis são criadas e manipuladas na memória RAM.
- Memória EEPROM: Memória utilizada para guardar informações de longo prazo, como dados fixos e *strings*.

Até versões mais recentes do FreeRTOS, os objetos do *kernel* como filas, semáforos e tarefas só podiam ser alocados dinamicamente, em tempo de execução. Dessa forma, o sistema aloca memoria RAM a cada vez que um objeto é criado e libera a cada vez que um objeto é deletado. A partir da versão 9.0.0 do sistema, é possível alocar os objetos dinamicamente em tempo de execução ou estaticamente, no tempo de compilação.

No FreeRTOS, alocações dinâmicas de memória podem ser realizadas utilizando as funções *malloc()* e *free()* da linguagem C. Como esse meio de alocação de memória pode não ser adequado para todos os cenários, a distribuição oficial oferece cinco esquemas de gerenciamento de memória, que dizem respeito a que políticas se utilizar ao alocar memória para criação de um novo objeto e a o que fazer com a memória liberada após a deleção de um objeto.

Os cinco esquemas de alocação se diferem em aspectos como de que forma é organizado o espaço ocupado pela *heap* de memória e o que fazem com o espaço livre que surge quando um objeto é excluído. A *heap 1*, como assim é chamada, é extremamente simples, ela apenas implementa funções de alocação, sem liberação de espaço ocupado. A *heap 2* implementa liberação de memória e utiliza um algoritmo para garantir que os blocos de memória escolhidos para serem usados tenham o tamanho mais próximo possível do requisitado pela alocação. A *heap 3*, implementa as funções de alocação e liberação de memória da linguagem C (*malloc()* e *free()*). A *heap 4* é similar ao segundo mas implementa a combinação de blocos livres adjacentes, o que diminui a fragmentação da memória em pedaços menores. Todos os esquemas apresentados até agora, com exceção do terceiro, alocam estaticamente um vetor contínuo na memória para usar como *heap*. Isso é o que os difere da *heap 5*, que apesar de funcionar de maneira semelhante à *heap 4*, mapeia previamente as regiões da memoria que serão utilizadas, dispensando assim, a necessidade delas serem contínuas.

No *port* para o Arduino Uno, não existem opções de escolha, uma vez que o único esquema de gerenciamento de memória disponível é a *heap 3*, que utiliza as funções de alocação e liberação de memória do C (*malloc()* e *free()*, como dito anteriormente).

3.3 O ARDUINO UNO

Esta sessão apresenta algumas informações acerca do *hardware* utilizado no projeto, a placa microcontroladora Arduino Uno. Todos os dados foram retirados do site da fabricante oficial (Arduino AG, 2018).

[2] SRAM é a sigla para *Static Random Acess Memory. É equivalente a memória RAM, mas mantém os bits de dados na memória enquanto o dispositivo estiver ligado, diferente da RAM, que é periodicamente atualizada.*

O Arduino UNO é uma placa microcontroladora de *design* aberto. Ela possui 14 pinos de entrada e saída, onde diversos dispositivos podem ser conectados, como um sensor captando a temperatura ambiente para fazer algum tipo de processamento ou um braço robótico que está sendo controlado pelo programa executando no *hardware*. A placa opera numa voltagem de *5V* e pode ser alimentada por uma fonte, bateria ou por uma entrada USB tipo B, que também é por onde é feito o *upload* do código-fonte ao ligar a placa em um computador.

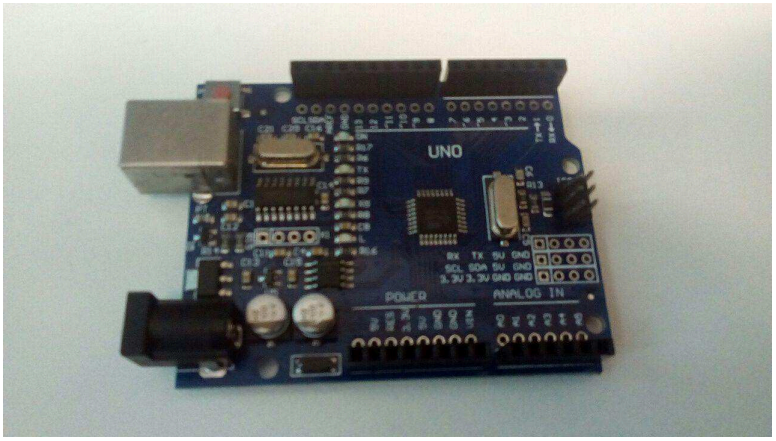

Figura 1 – A placa Arduino Uno R3.

Apesar de ser versátil e possibilitar a criação de uma infinidade de aplicações, o Arduino Uno é um componente de *hardware* bastante limitado. A tabela 1 mostra algumas das principais especificações da placa.

Microcontrolador	ATmega328P
Velocidade do clock	16 MHz
Memória Flash	32 KB , dos quais 0.5kb são usados pelo *bootloader*[3]
SRAM	2 KB

Tabela 1 – Algumas das especificações técnicas do Arduino UNO

3.3.1 Ambientes de desenvolvimento

A plataforma Arduino possui duas IDEs (*Integrated development environment* ou ambiente de desenvolvimento integrado) para o desenvolvimento e *upload* de programas para as placas microcontroladoras. Um dos ambientes é *online*, só necessitando a instalação de um *plug-in*. O outro é um programa que pode ser instalado em qualquer computador com MAC, Windows ou Linux. Ambos os ambientes possuem essencialmente os mesmos recursos e funcionalidades.

A linguagem usada para desenvolver programas para as placas Arduino é um conjunto de comandos das linguagens C e C++, além de comandos específicos da plataforma. Depois de pronto, o código é compilado e enviado para a placa através de uma conexão serial, estabelecida

[3] *Bootloader* é o trecho de código responsável pela inicialização do sistema operacional.

por um cabo USB tipo B entre ela e o dispositivo no qual a IDE está sendo executada. Uma vez que o código é totalmente copiado para a placa, ele começa a executar imediatamente e a saída serial pode ser monitorada através de uma funcionalidade do próprio ambiente.

Nos ambientes de desenvolvimento, existe um recurso que permite facilmente instalar uma série de bibliotecas que oferecem uma grande variedade de funcionalidades. O *port* do FreeRTOS está disponível como uma biblioteca neste recurso.

4 AVALIAÇÃO DE DESEMPENHO

Para medir o desempenho do sistema operacional aqui analisado, utilizou-se de alguns dos testes propostos por Krzysztof M. Sacha, em seu artigo *"Measuring the Real-Time Operating System Performance"* (Sacha, 1995). Os testes têm como objetivo obter medidas quantitativas de algumas das principais características do sistema operacional, a fim de tornar possível avaliá-lo e torná-lo mais facilmente comparável com outros sistemas operacionais.

Segundo Sacha, sistemas operacionais em tempo real podem ser caracterizados por dois requisitos básicos: pontualidade e confiabilidade. Para satisfazer esses requisitos, são utilizados alguns princípios, tais como *multitasking*[1] e comunicação eficiente entre tarefas. Para analisar a utilização desses princípios como garantia dos requisitos do sistema operacional, dois testes foram realizados: O primeiro mede a velocidade das trocas de contexto entre tarefas, o segundo mede a velocidade de transferência de dados entre tarefas. Ambos os testes são descritos a seguir.

4.1 LATÊNCIA MÉDIA DA TROCA DE CONTEXTO ENTRE TAREFAS

O tempo consumido pela troca de contexto entre duas tarefas é um *overhead*[2] naturalmente produzido por qualquer sistema operacional que funcione com *multitasking*. Além disso, este tempo está presente nos meios de comunicação entre tarefas, uma vez que é necessária a troca de contexto entre a tarefa que envia a informação e a que recebe. Portanto, este tempo é uma medida importante, que ajuda a caracterizar o desempenho do sistema operacional como um todo (Sacha, 1995).

Descrição do Teste: Nesse teste, duas tarefas executam simultaneamente: uma principal e uma secundária. Primeiro, a tarefa principal lê o tempo atual[3] e executa uma quantidade determinada de iterações num laço vazio. Então, ela lê o tempo atual novamente, calcula a diferença e divide pela quantidade de iterações. O resultado é a duração de tempo de uma iteração do laço na tarefa principal. Depois disso, a tarefa principal lê o tempo novamente e então executa um laço com o mesmo número de iterações do anterior, mas dessa vez, a cada iteração, ela entrega a CPU para a tarefa secundária. A tarefa secundária, por sua vez, tem como única função devolver a CPU para a tarefa principal. Esse processo continua até que o laço da tarefa principal termine. Quando isso ocorre, a tarefa principal lê o tempo mais uma vez e então calcula a diferença do tempo inicial com o atual, dividido pelo número de iterações (para se obter o tempo por iteração) dividido por dois (por serem duas tarefas, consideradas idênticas). Esse é o tempo de uma iteração do laço somado ao tempo da troca de contexto. Por fim, para se

[1] Quando mais de uma tarefa é realizada ao mesmo tempo.

[2] Excesso de processamento necessário para executar determinada tarefa

[3] A leitura do tempo é feita através da chamada da função *micros()*, segundo a documentação da API do Arduino, em placas com microcontroladores de 16MHz, como é o caso do Arduino Uno, a função tem resolução de 4 microssegundos.)

obter o tempo da troca de contexto, basta subtrair o tempo da iteração no laço da tarefa principal, calculado anteriormente.

```
void vMainTask(){
  volatile uint32_t count = LOOP_COUNT;
  uint32_t t1, t2;
  float tx, ty, TS;

  t1 = micros();
  do{
  }while(count--);
  t2 = micros();
  tx = ((float)t2-t1)/(float)LOOP_COUNT;
  count = LOOP_COUNT;
  t1 = micros();
  do{
    taskYIELD();
  }while(count--);
  t2 = micros();
  ty = (t2-t1)/ (float)LOOP_COUNT/2.0;
  TS = ty-tx;
  Serial.print(TS);
  vTaskDelete(NULL);
}
```

```
void vInterweavingTask(){
  do{
    taskYIELD();
  }while(1);
}
```

Figura 2 – Teste para troca de contexto no FreeRTOS.

Para evitar que o tempo medido nesse teste abranja outras atividades do sistema operacional, tal como execução periódica de certas tarefas, elas foram criadas com o maior nível de prioridade possível, sendo que a tarefa principal tem prioridade maior que a secundária, para que a secundária não execute até o momento que a principal lhe entregue a CPU.

4.2 LATÊNCIA MÉDIA DA TROCA DE MENSAGENS ENTRE TAREFAS

A maioria das ferramentas para comunicação entre tarefas podem ser divididas em duas categorias: síncronas e assíncronas. O FreeRTOS dispõe de dois mecanismos principais para esse fim: Filas, que podem ser usadas como *buffers* assíncronos e notificações de tarefas, um mecanismo síncrono que permite comunicação direta entre duas tarefas. Como os dois tipos de ferramenta possuem características e aplicações distintas, pode-se tornar difícil fazer um teste quantitativo das duas. Para contornar esse problema, propõe-se um teste constituído da troca de pares de mensagens entre duas tarefas, que pode ser facilmente implementado com qualquer ferramenta. O tempo a ser medido no seguinte teste corresponde à toda a operação: O envio da mensagem para a primeira tarefa, a troca de contexto e o recebimento da mensagem pela segunda tarefa. Segundo Sacha (1995), esta é a menor porção da operação que pode ser observada na vida real.

Descrição do Teste: Duas tarefas fazem parte do teste, executando simultaneamente: Uma tarefa cliente e uma tarefa servidor. A tarefa cliente lê o tempo atual e então faz uma quantia predeterminada de trocas de mensagens com a tarefa servidor, que simplesmente recebe a mensagem do cliente e responde. Depois que as trocas de mensagem acabam, a tarefa cliente

lê novamente o tempo, calcula a diferença com o tempo inicial, divide pelo número de iterações e então divide por dois, obtendo o tempo de apenas uma troca de mensagens.

Filas

```
void vClient(){
  volatile uint32_t count;
  uint32_t t1, t2;
  float TM;
  count = LOOP_COUNT;
  t1 = micros();
  do{
    xQueueSend(fila1, message, portMAX_DELAY);
    xQueueReceive(fila2, message, portMAX_DELAY);
  }while(count--);
  t2 = micros();
  TM = (t2-t1)/(double)LOOP_COUNT/2.;
  Serial.println(TM);
  vTaskDelete(NULL);
}
```

```
void vServer(){
  do{
    xQueueReceive(fila1, message, portMAX_DELAY);
    xQueueSend(fila2, message, portMAX_DELAY);
  }while(1);
}
```

Notificações de Tarefas

```
void vClient(){
  volatile uint32_t count;
  uint32_t t1, t2, val;
  float TM;
  count = LOOP_COUNT;
  t1 = micros();
  do{
    xTaskNotify(tServer, NUM ,eSetValueWithOverwrite);
    xTaskNotifyWait(0, 0, &val, portMAX_DELAY);
  }while(count--);
  t2 = micros();
  TM = (t2-t1)/(double)LOOP_COUNT/2.;
  Serial.println(TM);
  vTaskDelete(NULL);
}
```

```
void vServer(){
  uint32_t val;
  do{
    xTaskNotifyWait(0, 0, &val, portMAX_DELAY);
    xTaskNotify(tClient, NUM ,eSetValueWithOverwrite);
  }while(1);
  vTaskDelete(NULL);
}
```

Figura 3 – Teste para comunicação entre tarefas usando filas e notificações de tarefas no FreeRTOS.

5 RESULTADOS

Esse capítulo contém os resultados das análises do gerenciador de memória, dos recursos de tarefa e dos recursos de fila da implementação do FreeRTOS para Arduino Uno, assim como os resultados dos testes propostos no capítulo 4.

5.1 DOCUMENTAÇÃO AUXILIAR

Durante a análise do código fonte do *port* para Arduino Uno do FreeRTOS, documentou-se o funcionamento de várias funções bem como o tamanho e composição de várias estruturas que compõe o sistema. Essa documentação foi então disponibilizada publicamente através da plataforma Zenodo[1] e pode ser acessada através das referências do presente trabalho (Chabatura; Tironi Fassini, 2018).

5.2 GERENCIAMENTO DE MEMÓRIA

Como mencionado anteriormente, o *port* para Arduino Uno do FreeRTOS possui apenas uma opção de gerenciamento de memória: as funções da biblioteca padrão *malloc()* e *free()*. Para entender as implicações deste esquema de gerenciamento de memória para a escalabilidade e desempenho de aplicações, é necessário primeiro entender como ele funciona.

O Arduino Uno utiliza uma biblioteca de funções padrões da linguagem C adaptadas para dispositivos AVR, chamada *AVR Libc*. Todas as informações sobre o funcionamento do gerenciamento de memória e sobre como é organizada a memória RAM foram retiradas dos manuais da biblioteca (AVR-Libc, 2016).

5.2.1 Regiões da Memória RAM

A maioria dos dispositivos AVR, assim como o Arduino Uno, possuem uma quantidade bastante limitada de memória RAM. Essa memória deve ser divida entre as diversas sessões de memória, a *heap*, a pilha e o alocador de memória dinâmica. As principais áreas da memória são descritas a seguir:

- ***.data*:** Sessão onde ficam as variáveis inicializadas e/ou com dados estáticos.
- ***.bss*:** Sessão onde ficam as variáveis globais ou estáticas não inicializadas.
- ***heap*:** Espaço ocupado pelas alocações dinâmicas de memória.
- ***stack* (pilha):** Espaço usado para chamar sub-rotinas e armazenar variáveis locais.

[1] Zenodo é um repósito de propósito geral para trabalhos técnicos e científicos. Ele provê DOI (*Digital Object Identifier* ou identificador de objeto digital) para as publicações nele armazenadas, um código que identifica um objeto digital de maneira única e persistente.

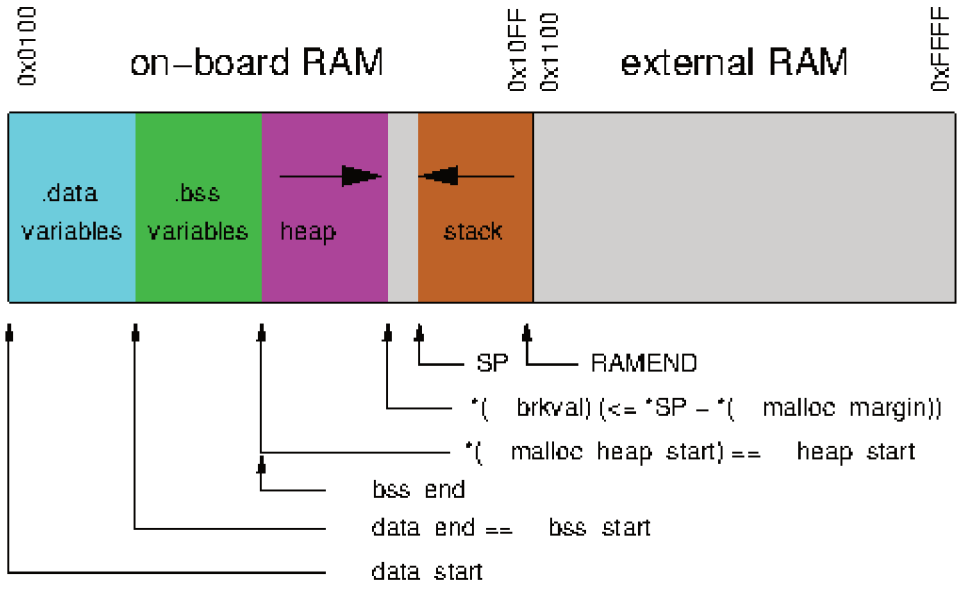

Figura 4 – Mapa do layout da memória RAM, adotado de (AVR-Libc, 2016).

No Arduino Uno, assim como em outras arquiteturas AVR, não há recursos de *hardware* para gerenciamento de memória que poderiam ajudar a evitar que sessões da memória se sobrescrevam. O *layout* padrão de memórias RAM é de colocar a sessão *.data* no início da memória RAM, seguido da sessão *.bss*. A pilha começa a crescer do topo (endereço mais alto) da memória RAM e a *heap* é colocada após o fim da sessão *.bss* e cresce em direção contrária da pilha. Dessa forma, não há forma da memória dinâmica colidir com as variáveis da RAM, porém, ainda há um risco da pilha e da *heap* colidirem. Isso pode acontecer caso haja alta demanda por alocações dinâmicas ou espaço de pilha, seja por variáveis locais ou chamadas recursivas de funções. Esse *layout* é ilustrado na figura 4.

5.2.2 Funcionamento da Alocação Dinâmica

O Gerenciador de Memória utilizado pela *avr-libc* não mantém nenhuma estrutura de dados separada para manter o controle dos blocos livres de memória. Para isso, ele implementa uma lista encadeada de blocos livres na própria *heap*. Cada bloco de memória livre, é precedido de 4 *bytes* contendo o ponteiro para o próximo bloco livre e o tamanho do bloco, respectivamente. Além disso, o endereço do espaço alocado retornado por uma chamada de *malloc()* é precedido de dois *bytes* que indicam o tamanho do bloco ocupado.

Ao receber uma chamada de alocação de memória (*malloc()*), a lista encadeada é percorrida em busca de um bloco que atenda à solicitação de tamanho perfeitamente. Caso esse bloco exista, ele será desconectado da lista e retornado para quem chamou a função. Caso contrário, o de tamanho mais próximo à requisição é utilizado. Nesse caso, esse bloco é dividido em dois: Um é retornado à quem chamou a função, o outro permanece na lista. Caso o bloco encontrado seja apenas dois *bytes* maior que o requisitado, esses dois *bytes* serão inclusos na requisição, já que não seria possível criar um novo bloco livre com eles. Caso não hajam blocos livres capazes

de lidar com a solicitação, faz-se uma tentativa de estender a *heap*. Se não houver memória suficiente para atender à solicitação, *NULL* é retornado para quem chamou a função.

Ao receber uma chamada de liberação de memória (*free()*), são realizadas tentativas de juntar o bloco sendo liberado com possíveis blocos livres adjacentes. Isso é realizado como tentativa de amenizar o problema de fragmentação de memória: Espaços livres que se criam no meio da *heap* depois de algumas alocações e liberações e acabam não podendo ser utilizados por não satisfazerem o tamanho das novas alocações, causando desperdício de memória. Quando o bloco mais acima na *heap* é desalocado, o tamanho da *heap* também diminui.

Ao receber uma chamada de realocação de memória (*realloc()*), primeiro determina-se se o bloco vai aumentar ou diminuir de tamanho. Caso for diminuir, o bloco é dividido e a parte final é passada para a função de liberação caso tenha tamanho suficiente para virar um bloco livre. Se não, a solicitação é ignorada. Caso o bloco for aumentar de tamanho, checa-se se é possível fazer isso localmente primeiro. Se sim, um ponteiro é retornado com o endereço do bloco estendido, sem se fazer nenhuma cópia. Se não, verifica-se a possibilidade de usar um dos blocos livres da lista para atender a requisição, copiando o conteúdo e liberando o espaço antigo. Se isso não for possível, uma chamada de alocação é feita para o novo espaço, o conteúdo é copiado e o espaço antigo é liberado. Caso o bloco a ser estendido seja o topo da *heap*, é feita uma tentativa de extensão da *heap* para evitar a necessidade de se copiar o conteúdo. Caso a *heap* precise aumentar de tamanho e não possa durante algum momento do processo, toda a solicitação falha.

5.2.3 Como o FreeRTOS lida com alocações dinâmicas

Na implementação do FreeRTOS, a função *malloc()*, da biblioteca *AVR Libc*, é encapsulada por outra função, chamada *pvPortMalloc()*, presente no arquivo *heap_3.c* da distribuição para Arduino Uno. Quando acontece uma alocação dinâmica de memória, é necessário garantir que a operação seja segura (*thread-safe*[2]), ou seja, não ocorram acessos indevidos à memória antes do processo de alocação acabar. Para implementar essa segurança, o FreeRTOS cria uma sessão crítica, suspendendo o escalonador antes do início da alocação e o retomando apenas quando o processo estiver terminado.

A suspensão do escalonador é realizada através de uma chamada da função *vTaskSuspendAll()*, presente no arquivo *tasks.c*. Essa função simplesmente atribui uma variável global (*uxSchedulerSuspended*), indicando que o escalonador está suspenso. Enquanto o escalonador está suspenso, trocas de contexto não ocorrem, porém as interrupções não são desabilitadas. Os *ticks*[3] ficarão pendentes até que o escalonador seja retomado através de uma chamada à função *xTaskResumeAll()*. Depois que o escalonador é suspenso, uma chamada à função *malloc()* é

[2] Uma aplicação é considerada *thread-safe* quando não causa problemas ao ser executada num cenário de múltiplas *threads* (ou tarefas, no caso do *FreeRTOS*).

[3] Cada ciclo de processamento do microcontrolador.

realizada. O retorno da chamada é salvo e será o endereço para o início do bloco de memória alocado em caso de sucesso e *NULL* caso contrário.

Depois que a alocação é feita, o funcionamento do escalonador é retomado através de uma chamada da função *xTaskResumeAll()*, presente no arquivo *tasks.c*. Primeiramente, a função entra numa região crítica, desabilitando globalmente as interrupções através de uma chamada da função *taskENTER_CRITICAL()*. Então, ela atribui a mesma variável global da função que suspende o escalonador (*uxSchedulerSuspended*), mas agora indicando que o escalonador não está mais suspenso. É possível que enquanto o escalonador estava suspenso, alguma tarefa tenha sido removido de uma lista de evento[4] através de uma rotina de serviço de interrupção (ISR). Se esse for o caso, essa tarefa terá sido adicionada à lista de tarefas que estão pendentes para ficarem prontas (*xPendingReadyList*). Uma vez que o escalonador é resumido, estas tarefas devem ser movidas para a lista de pronto apropriada. Caso alguma dessas tarefas tenha prioridade maior que a que está sendo executada atualmente, uma troca de contexto é marcada como pendente. Caso hajam *ticks* pendentes do tempo em que o escalonador estava suspenso, eles devem ser processados agora, para garantir que não hajam erros no contador de *ticks* e tarefas atrasadas sejam continuadas no tempo correto. Durante o processamento dos *ticks* pendentes, é possível que uma troca de contexto seja marcada como pendente também, uma vez que o processamento de um *tick* pode desbloquear tarefas com maior prioridade que a atual. Depois de processados os *ticks* pendentes, caso haja uma troca de contexto pendente, ela é realizada contanto que o uso de preempção esteja habilitado (*configUSE_PREEMPTION* não seja 0). Por fim, a função sai da sessão crítica com uma chamada de *taskEXIT_CRITICAL()*.

Após concluído o processo de resumir o escalonador, a função *pvPortmalloc()* analisa o retorno da função *malloc()*, retornando o endereço para o bloco de memória alocado em caso de sucesso ou NULL caso contrário. A implementação da função *pvPortMalloc()* também permite que, caso o retorno da função *malloc()* seja *NULL*, uma função *hook*[5] seja executada. Para isso, a definição da constante *configUSE_MALLOC_FAILED_HOOK* deve ser verdadeira e a função (*vApplicationMallocFailedHook()*) deve ter sido previamente declarada pelo desenvolvedor da aplicação.

Ao se analisar o comportamento da função *pvPortMalloc()*, pode-se supor que quanto maior a duração da execução da função *malloc()*, maior será a quantidade de processamento e, consequentemente, tempo necessário para resumir o funcionamento do escalonador. Esse comportamento é esperado por que, no geral, quanto maior o tempo em que o escalonador fica suspenso, maior a probabilidade de existir um maior número de *ticks* suspensos e tarefas pendentes para serem trocadas de listas quando o escalonador for resumido. Como visto na sessão 5.2.2, o processo de alocação dinâmica não é simples, tornando a operação como um todo bastante custosa. Porém, espera-se que o custo de tempo varie bastante de acordo com uma série de fatores tais como tamanho do espaço a ser alocado, quantidade de alocações dinâmicas

[4] Listas onde tarefas esperam por um evento ocorrer. Geralmente esse evento está ligado a uma rotina de serviço de interrupção.

[5] Funções *hook* permitem adicionar funcionalidades ao sistema em resposta a determinados eventos.

anteriores, número de tarefas executando e o estado atual do sistema. Para maiores informações sobre todas as funções citadas nesse capítulo, referir-se à documentação complementar (Chabatura; Tironi Fassini, 2018).

5.3 TAREFAS

Como mencionado anteriormente, cada tarefa possui na memória uma pilha e um bloco de controle (TCB). O tamanho da pilha da tarefa é determinado no momento da sua criação e varia do mínimo de 85 *bytes* (difinido pela constante *configMINIMAL_STACK_SIZE* no arquivo *FreeRTOSConfig.h*) até o máximo possível de ser alocado sem que ocorra uma *heap overflow*[6].

Para melhor compreensão do espaço ocupado por cada tarefa na memória, é preciso analisar os elementos que compõe os blocos de controle de cada tarefa, além de como ocorre o processo de criação das tarefas.

5.3.1 Estrutura do TCB

A tabela 2 mostra os principais campos da estrutura de bloco de controle (TCB) de cada tarefa, definida como *TCB_t* no arquivo *tasks.c*. Por motivos de clareza e simplicidade, alguns campos referentes à recursos ausentes no *port* para Arduino Uno (como proteção de memória), recursos de *debugging* e/ou que dizem respeito ao uso de bibliotecas de terceiros foram deixados de fora dessa análise. Maiores detalhes sobre todos os campos do TCB e as estruturas que definem o tipo de cada campo podem ser encontradas na documentação auxiliar, presente nas referências desse trabalho (Chabatura; Tironi Fassini, 2018).

Campo	Tipo	Tamanho (bytes)	Obrigatório	Habilitado por Padrão
pxTopOfStack	*stackType_t**	2	**Sim**	-
xGenericListItem	*ListItem_t*	10	**Sim**	-
xEventListItem	*ListItem_t*	10	**Sim**	-
uxPriority	*UBaseType_t*	1	**Sim**	-
pxStack	*StackType_t**	2	**Sim**	-
pcTaskName	*char[]*	8	**Não**	**Sim**
uxBasePriority	*UBaseType_t*	1	**Não**	**Sim**
uxMutexesHeld	*UBaseType_t*	1	**Não**	**Sim**
ulNotifiedValue	*uint32_t*	4	**Não**	**Sim**
eNotifyState	*eNotifyValue*	2	**Não**	**Sim**

Tabela 2 – Principais campos do TCB de cada tarefa.

A seguir, descreve-se cada um dos principais campos que compõe o TCB:

- ***pxTopOfStack***: Ponteiro para a localização do último item colocado na pilha da tarefa.

[6] Quando há uma tentativa de fazer uma alocação maior que o espaço disponível na *heap*, ocorre um transbordamento da *heap*, ou *heap overflow*.

- ***xGenericListItem***: A lista referenciada por este item denota o estado atual da tarefa (pronta, bloqueada, suspensa).
- ***xEventListItem***: Caso a tarefa esteja em uma lista de eventos, ela será referenciada por esse item.
- ***uxPriority***: Prioridade da tarefa. O valor mínimo é 0.
- ***pxStack***: Ponteiro para o começo da pilha da tarefa.
- ***pcTaskName***: Nome descritivo da tarefa, para facilitar em processos de *debugging*.
- ***uxBasePriority***: Caso a funcionalidade de *mutexes* esteja ativada, guarda a última prioridade atribuída à tarefa. Esse valor é usado pelo mecanismo de herança de prioridades.
- ***uxMutexesHeld***: Caso a funcionalidade de *mutexes* esteja ativada, guarda a quantidade de *mutexes* segurados pela tarefa.
- ***ulNotifiedValue***: Caso a funcionalidade de notificações de tarefa esteja ativada, guarda o valor de notificação da tarefa, um inteiro de 32 *bits*.
- ***eNotifyState***: Guarda o estado de notificação da tarefa, pode ser um dentre os seguintes valores: (*eNotWaitingNotification*, *eWaitingNotification*, *eNotified*).

A partir da análise dos campos listados acima, é possível determinar que o tamanho do TCB por padrão é de 41 *bytes*, o máximo possível sem habilitar nenhuma funcionalidade de *debugging* e/ou recursos de terceiros ou aumentar o tamanho máximo do nome da tarefa. Além disso, é possível verificar que o tamanho mínimo do TCB é de 27 *bytes*. Esses valores não levam em consideração os 2 *bytes* provenientes da alocação dinâmica, apenas o tamanho da estrutura.

A sessão 5.3.2 descreve como cada TCB é criando enquanto as sessões 5.2.2 e 5.2.3 contém mais informações sobre alocações dinâmicas no FreeRTOS e suas consequências.

5.3.2 Criação de Tarefas

A seguir, descreve-se o funcionamento da função responsável por criar as tarefas (*xTaskGenericCreate*, no arquivo *tasks.c*). Por motivos de clareza e simplicidade, omitiu-se dessa descrição as chamadas de macros de *tracing*, funcionalidades de *debugging* e recursos não disponíveis no *port* do FreeRTOS para o Arduino Uno. Para maiores detalhes sobre todas funções, referir-se à documentação complementar (Chabatura; Tironi Fassini, 2018).

A primeira coisa que a função faz é alocar o espaço necessário para o TCB e para a pilha da nova tarefa, através de uma chamada da função *prvAllocateTCBAndStack()* (também no arquivo *tasks.c*). Esta função determina, de acordo com a arquitetura, se a pilha cresce dos endereços mais baixos de memória para os mais altos, ou o contrário. Caso a pilha cresça dos endereços mais baixos para os mais altos, o TCB é alocado antes da pilha. Caso a pilha cresça dos endereços mais altos para os mais baixos, o TCB é alocado depois da pilha. Essa checagem garante que a pilha nunca cresça na direção do TCB, de modo a poder sobrescrevê-lo. No caso

do *port* para Arduino Uno, a pilha cresce em direção aos endereços mais baixos, sendo, portanto, alocada antes do TCB. Tanto a alocação da pilha, quanto do TCB são feitas através da função *pvPortMalloc()*, que encapsula a função *malloc()*, ambas descritas nas sessões 5.2.2 e 5.2.3, respectivamente. Como consequência do uso dessas funções, a pilha e o TCB ocupam 2 *bytes* a mais na memória cada, provenientes das alocações dinâmicas. Caso as alocações sejam bem sucedidas, o endereço do começo da pilha é atribuído ao campo *pxStack* do TCB e o endereço do TCB criado é retornado. Caso contrário, qualquer espaço utilizado é liberado, e a função retorna *NULL*.

Depois de alocados a pilha e o TCB, o endereço para o topo da pilha é calculado dependendo da direção em que ela cresce. Então, os demais campos do TCB são inicializados através de uma chamada à função *prvInitialiseTCBVariables()*. Esta função inicializa todas os campos do TCB não inicializados até agora, tais como nome, prioridade, os itens de lista, entre outros. Depois disso, a pilha da tarefa é inicializada (através da função *pxPortInitialiseStack()*) de modo a parecer como se a tarefa já estivesse sendo executada, porém tivesse sido interrompida pelo escalonador. O início da função da tarefa é atribuído ao endereço de retorno. Caso tenha sido informado um endereço para armazenar o *handle* da tarefa, ele é copiado para tal.

Depois disso, as estruturas globais sãos atualizadas para se adequarem a criação da nova tarefa. Para isso, é necessário desabilitar as interrupções através de uma chamada de *taskENTER_CRITICAL()*, para que elas não acessem as listas de tarefas durante as atualizações. Se essa tarefa está sendo a primeira a ser criada, é necessário inicializar as listas de tarefas, isso é feito através de uma chamada à função *prvInitialiseTaskLists()*. Essa tarefa é responsável por inicializar todas as listas globais responsáveis pelo controle das tarefas atrasadas, pendentes, prontas, entre outras. Caso não haja outra tarefa marcada como atual ou o escalonador ainda não esteja sendo executado e a tarefa sendo criada tenha a maior prioridade dentre todas até agora, ela é atribuída como tarefa atual.

Se a função chegou até aqui sem erros, é atribuído *pdPASS* à variável de retorno da função, indicando sucesso. Além disso ela é colocada na lista de pronto e a função sai da região crítica com uma chamada de *taskEXIT_CRITICAL()*. Caso o escalonador esteja rodando enquanto a tarefa é criada, avalia-se se é necessário fazer uma troca de contexto. Isso só ocorre se a tarefa que acabou de ser criada tem maior prioridade que a atualmente rodando.

5.3.3 Máximo alocável de tarefas

Sabendo exatamente quanto cada tarefa ocupa de espaço na memória, como descrito nas sessões anteriores, é possível determinar com certa precisão o mínimo de espaço necessário para execução de determinada aplicação. Porém, para ilustrar as capacidades e limitações do *port* do FreeRTOS no Arduino Uno, testou-se o máximo de tarefas que poderiam ser alocadas, além do tamanho máximo de pilha alocável por tarefa quando determinado número de tarefas é criado. Para determinar esses valores, testou-se o maior tamanho de pilha possível para as

tarefas alocadas, sem que ocorra uma *heap overflow*. No *port* para Arduino Uno do FreeRTOS, quando ocorre uma *heap overflow*, o LED da placa pisca num ciclo de 100 milissegundos (Stevens, 2018), possibilitando dessa forma, determinar exatamente o máximo de tarefas que é possível alocar e o tamanho máximo da pilha de cada tarefa. O teste foi realizado utilizando as configurações padrões do *port* do FreeRTOS para Arduino Uno disponibilizado nas bibliotecas da plataforma Arduino.

Qtd. Tarefas	Tamanho Máximo da pilha (bytes)	Pilhas + TCB's (*bytes*)
1	1297	1338
2	626	1334
3	402	1329
4	290	1324
5	223	1320
6	178	1314
7	146	1309
8	122	1304
9	104	1305
10	89	1300

Tabela 3 – Tamanho máximo de pilha alocável por tarefa de acordo com a quantidade de tarefas.

A tabela 3 relaciona a quantidade de tarefas criadas com o máximo de tamanho de pilha que é possível de ser alocado. Os valores contidos na terceira coluna da tabela não levam em conta o *overhead* produzido pela alocação dinâmica de cada pilha e TCB. Esse teste foi realizado utilizando as configurações padrões do *port* do FreeRTOS para Arduino Uno.

5.4 FILAS

Como mencionado anteriormente, as filas são um mecanismo amplamente utilizado pelo sistema operacional, servindo como *buffers* de dados e participando no processo de comunicação entre tarefas ou de tarefas com serviços de interrupção. No FreeRTOS, a estrutura de filas também é utilizada para implementar a funcionalidade de *mutexes*. Por questões de clareza, no presente trabalho restringe-se às funcionalidades das estruturas para implementação de filas. Para maiores informações sobre a utilização da estrutura para a implementação de *mutexes*, referir-se à documentação auxiliar presente nas referências desse trabalho (Chabatura; Tironi Fassini, 2018).

Nessa sessão, serão descritos os principais campos que compõe a estrutura de filas, assim como o seu processo de criação.

5.4.1 Estrutura da fila

A tabela 4 mostra os principais campos da estrutura de fila, definida como *Queue_t*, no arquivo *queue.c*. Novamente, por questões de clareza, omitiram-se campos referentes a

funcionalidades de *debugging*. Maiores informações sobre todos os campos da fila podem ser encontrados na documentação auxiliar (Chabatura; Tironi Fassini, 2018).

Campo	Tipo	Tamanho (bytes)	Obrigatório	Habilitado por Padrão
pcHead	*int8_t**	2	**Sim**	-
pcTail	*int8_t**	2	**Sim**	-
pcWriteTo	*int8_t**	2	**Sim**	-
pcReadFrom/ uxRecursiveCallCount	*int8_t* / UBaseType_t*	2	**Sim**	-
xTasksWaitingToSend	*List_t*	9	**Sim**	-
xTasksWaitingToReceive	*List_t*	9	**Sim**	-
uxMessagesWaiting	*UBaseType_t*	1	**Sim**	-
uxLength	*UBaseType_t*	1	**Sim**	-
uxItemSize	*UBaseType_t*	1	**Sim**	-
xRxLock	*UBaseType_t*	1	**Sim**	-
xTxLock	UBaseType_t	1	**Sim**	-
pxQueueSetContainer	QueueDefinition*	2	**Não**	**Não**

Tabela 4 – Principais campos da estrutura de fila.

A seguir, descrevem-se os principais campos que compõe a estrutura das filas:

- ***pcHead***: Ponteiro para o ínicio da área de armazenamento da fila.
- ***pcTail***: Ponteiro para o *byte* no final da área de armazenamento da fila. Um *byte* a mais que o necessário é alocado para armazenar os itens da fila, ele é usado como marcador.
- ***pcWriteTo***: Ponteiro para o próximo espaço livre na área de armazenamento.
- ***pcReadFrom/uxRecursiveCallCount***: Como estes dois itens nunca coexistem, usa-se uma *union*[7] para economizar RAM. Pode ser um ponteiro para o último lugar de onde um item enfileirado foi lido ou um contador para o número de vezes que um *mutex* recursivo foi recursivamente pego, quando a estrutura é usada como um *mutex*.
- ***xTasksWaitingToSend***: Lista das tarefas que estão bloqueadas esperando para escrever nessa fila. As tarefas são armazenadas em ordem de prioridade.
- ***xTasksWaitingToReceive***: Lista das tarefas que estão bloqueadas esperando para ler dessa fila. As tarefas são armazenadas em ordem de prioridade.
- ***uxMessagesWaiting***: Número de itens atualmente na fila.
- ***uxLength***: Tamanho da fila em capacidade total de itens, não *bytes*.
- ***uxItemSize***: Tamanho de cada item da fila em *bytes*.
- ***xRxLock***: Armazena o número de itens removidos da fila enquanto a fila estava bloqueada. Quando a fila não está bloqueada, vale *queueUNLOCKED*, que é definida como -1.
- ***xTxLock***: Armazena o número de itens adicionados à fila enquanto a fila estava bloqueada. Quando a fila não está bloqueada, vale *queueUNLOCKED*, que é definida como -1.

[7] *Unions* são tipos de dados especiais da linguagem C que permitem armanezar diferentes tipos de dados no mesmo endereço de memória. Apenas um dos elementos da *union* pode existir em dado instante de tempo.

- ***pxQueueSetContainer***: Caso o recurso de *"queue sets"* esteja ativado, aponta para o conjunto a qual essa fila pertence. *"Queue sets"* é um recurso do FreeRTOS para que uma tarefa desbloqueie ao conseguir ler de mais de uma fila, por exemplo.

A partir da análise dos campos listados acima e seus respectivos tamanhos, pode-se determinar que o tamanho da estrutura de uma fila por padrão é de 31 *bytes*. O tamanho máximo da estrutura é de 33 *bytes*, quando a funcionalidade de conjuntos de fila (*queue sets*) está habilitada. Este tamanho só diz respeito a estrutura de controle da fila, o tamanho da área de armazenamento pode ser calculando multiplicando a capacidade máxima da fila (*uxLenght*) pelo tamanho de cada item (*uxItemSize*) e somando 1 byte, alocado para ser usado como marcador. Como descrito na sessão 5.4.2, apenas uma alocação dinâmica é realizada para criar a estrutura da fila e sua área de armazenamento, reduzindo o espaço consumido pelo gerenciador de memória. Maiores informações sobre alocações dinâmicas no FreeRTOS podem ser encontradas nas sessões 5.2.2 e 5.2.3.

5.4.2 Criação de Filas

A seguir, descreve-se o funcionamento da função responsável pela criação das filas (*xQueueGenericCreate*, no arquivo *queue.c*). Novamente, por motivos de simplicidade e clareza, se omitirá os detalhes condizentes à macros de *tracing* e funcionalidades de *debugging*. Para maiores detalhes sobre todas as funções envolvidas no processo, referir-se a documentação complementar, presente nas referências desse trabalho (Chabatura; Tironi Fassini, 2018).

As filas são compostas por dois elementos localizados sequencialmente na memória: A estrutura da fila em si e a área de armazenamento, onde ficam guardados os itens da fila. O tamanho da área de armazenamento é determinado pelo número de itens que a fila suporta multiplicado pelo tamanho de cada item. Então, o espaço total ocupado pela fila (estrutura mais a área de armazenamento) é alocado sequencialmente na memória, através de uma chamada de *pvPortMalloc*.

Depois da alocação do espaço ocupado pela fila, calcula-se o valor da variável *pcHead*, que passa a apontar para o início da área de armazenamento. Após, os campos da estrutura da fila *uxQueueLength* e *uxItemSize* tem seus respectivos valores atribuídos e a fila é configurada para um estado inicial através de uma chamada da função *xQueueGenericReset*. Essa função é responsável por atribuir às outras variáveis da estrutura da fila seus valores padrão. Como ela pode ser chamada para uma fila já existente, a função entra na região crítica (chamando *taskENTER_CRITICAL()*) para fazer isso, pois tarefas podem ser desbloqueadas durante o processo, uma vez que se há uma tarefa esperando para escrever na fila, ela já pode ser desbloqueada, pois a fila ficará vazia. Depois que a chamada da função *xQueueGenericReset* acaba, o endereço (*handle*) da fila é retornado a quem chamou a função.

5.4.3 Máximo de filas por quantidades de tarefa

De maneira análoga ao teste para determinar a quantidade máxima de tarefas possíveis de serem criadas, realizou-se um teste para ilustrar a capacidade do *port* para Arduino Uno do FreeRTOS em relação à quantidade máxima de filas. Como o tamanho das filas varia de acordo com a sua capacidade máxima e com o tamanho de cada item, para o propósito do teste, fixou-se que cada fila conteria cinco itens de quatro *bytes* cada. O tamanho da pilha de cada tarefa criada é o mínimo possível (85 *bytes*, como visto no início da sessão 5.3). Do mesmo modo que no teste para determinar a quantidade máxima de tarefas alocáveis, utilizou-se do recurso de detecção de *heap overflow* do *port* do FreeRTOS para o Arduino Uno para determinar o máximo possível de filas alocáveis. O teste foi realizado utilizando as configurações padrões do *port* do FreeRTOS para Arduino Uno disponibilizado nas bibliotecas da plataforma Arduino.

Quantidade de Tarefas	Quantidade Máxima de Filas
1	22
2	19
3	17
4	15
5	12
6	10
7	7
8	5
9	3
10	0

Tabela 5 – Máximo de filas por quantidade de tarefas.

A tabela 5 relaciona a quantidade máxima de filas com cinco itens de quatro *bytes* que é possível criar de acordo com o número de tarefas existentes.

5.5 TESTES DE DESEMPENHO

Esta sessão apresenta os resultados provenientes dos testes para avaliação de desempenho descritos no capítulo 4.

5.5.1 Latência Média da Troca de Contexto Entre Tarefas

Número de Iterações (Amostras)	Tempo Médio da Troca de Contexto (µs)
10000	7.74
20000	7.74
30000	7.74

Tabela 6 – Latência média da troca de contexto do FreeRTOS no Arduino Uno.

A tabela 6 mostra o tempo médio de troca de contexto de acordo com o número de iterações do código teste descrito na sessão 4.1. A variação na quantidade de amostras não causou grande diferença no tempo médio das trocas de contexto, o que significa um resultado positivo em termos de previsibilidade e confiabilidade do sistema.

5.5.2 Latência Média da Troca de Mensagens Entre Tarefas

Número de Iterações (Amostras)	Tamanho da Mensagem (bytes)	Tempo de Transferência (µs)
100000	1	94.35
100000	10	103.29
100000	50	143.57
100000	100	193.73
100000	250	344.61
100000	500	338.52

Tabela 7 – A duração média da transferência de mensagens usando filas.

Número de Iterações (Amostras)	Tamanho da Mensagem (bytes)	Tempo de Transferência (µs)
100000	4	50.28

Tabela 8 – A duração média da transferência de mensagens usando notificações de tarefas.

As tabelas 7 e 8 mostram os tempos médios de envio de mensagens entre tarefas utilizando filas e notificações de tarefas respectivamente. Pôde-se comprovar a grande diferença de desempenho do recurso de notificações de tarefa em relação à comunicação usando filas, apesar de suas limitações. Outro ponto notável é que a correspondência entre o tamanho da mensagem e o tempo necessário para ela ser enviada utilizando filas, é fortemente não linear. Essa característica já havia sido salientada por Krzysztof Sacha, quando propôs o teste em seu trabalho (Sacha, 1995).

6 CONSIDERAÇÕES FINAIS

6.1 CONCLUSÃO

Alcançou-se com o presente trabalho os objetivos propostos de se criar uma análise dos recursos de gerenciamento de memória, tarefas e filas do sistema operacional em tempo real FreeRTOS na plataforma de *hardware* Arduino Uno. A descrição do funcionamento do gerenciador de memória possibilitou uma maior compreensão do impacto que esse esquema tem na escalabilidade e desempenho do sistema como um todo. Quanto aos recursos de filas e tarefas, pode-se observar através das estruturas de cada um e de como ocorrem suas instanciações na memória, quão escaláveis essas funcionalidades são num ambiente bastante limitado, como é o caso da maioria das placas microcontroladoras. Além disso, pôde-se através dos testes de latência de troca de contexto e de latência de troca de mensagens, obter algumas medições relevantes para se ter um panorama do desempenho geral do sistema operacional. Por fim, criou-se a documentação auxiliar descrevendo os tipos, estruturas e funções estudadas no decorrer do projeto. Espera-se que ela possa ajudar numa melhor compreensão de como o FreeRTOS e em especial o *port* para o Arduino Uno funcionam. A documentação está disponível nas referências desse trabalho (Chabatura; Tironi Fassini, 2018).

Por fim, o trabalho proporcionou um valioso aprendizado a respeito do quão limitado e desafiador é o cenário de sistemas em tempo real, em especial os projetados para sistemas embarcados. Além disso, toda a experiência proporcionada pelos processos envolvidos na análise e aplicação de testes de desempenho e escalabilidade de um sistema operacional foram de grande importância.

6.2 TRABALHOS FUTUROS

Como sugestões para trabalhos futuros, salienta-se que a versão do FreeRTOS utilizada nesse trabalho não foi a última a ser lançada. As versões mais novas apresentam recursos que podem proporcionar um melhor desempenho em alguns cenários, como é o caso do recurso de alocação estática de objetos do *kernel*, disponível a partir da versão $9.0.0$. Além disso, a partir da versão $10.0.0$ do sistema, foram implementadas as funcionalidades de *stream buffers* e *message buffers*. Esses recursos são primitivas de comunicação entre tarefas e de tarefas com serviços de interrupção otimizados para cenários onde há apenas uma tarefa (ou serviço de interrupção) lendo ou escrevendo.

REFERÊNCIAS

1 AG, Arduino. **Memory**. 2018. Disponível em: <https://www.arduino.cc/en/Tutorial/Memory>. Acesso em: 22 nov. 2018.

2 ARDUINO AG. **ARDUINO UNO REV3**. [S.l.: s.n.], 2018. Disponível em: <https://store.arduino.cc/arduino-uno-rev3>. Acesso em: 18/06/2018.

3 AVR-LIBC. **Memory Areas and Using malloc()**. 2016. Disponível em: <https://www.nongnu.org/avr-libc/user-manual/malloc.html>. Acesso em: 19 nov. 2018.

4 BARRY, Richard. Mastering the FreeRTOS Real Time Kernel-a Hands On Tutorial Guide. **Real Time Engineers Ltd**, 2016.

5 CHABATURA, Felipe; TIRONI FASSINI, Leonardo. **Descrições de Funções e Estruturas da Versão 8.2.3 do FreeRTOS para o Arduino Uno**. [S.l.], nov. 2018. DOI: 10.5281/zenodo.1580268. Disponível em: <https://doi.org/10.5281/zenodo.1580268>.

6 RAMMIG, Franz et al. Basic concepts of real time operating systems. In: HARDWARE-DEPENDENT Software. [S.l.]: Springer, 2009. p. 15–45.

7 SACHA, Krzysztof M. Measuring the real-time operating system performance. In: IEEE. REAL-TIME Systems, 1995. Proceedings., Seventh Euromicro Workshop on. [S.l.: s.n.], 1995. p. 34–40.

8 STEVENS, Phillip. **Arduino FreeRTOS Library**. [S.l.: s.n.], 2018. Disponível em: <https://github.com/feilipu/Arduino_FreeRTOS_Library>. Acesso em: 18/06/2018.